ISSN 2079-3316 PROGRAM SYSTEMS: THEORY AND APPLICATIONS no.3(34), 2017, pp. 61–85

N. S. Zhivchikova, Y. V. Shevchuk

Riak KV performance in sensor data storage application

Abstract. A sensor data storage system is an important part of data analysis systems. The duty of sensor data storage is to accept time series data from remote sources, store them and provide access to retrospective data on demand. As the number of sensors grows, the storage system scaling becomes a concern. In this article we experimentally evaluate Riak KV—a scalable distributed key-value data store as a backend for a sensor data storage system.

Key words and phrases: sensor data, write performance, distributed storage, time series, Riak, Erlang.

2010 *Mathematics Subject Classification*: 68M20

Introduction

The purpose of a sensor data storage is to store data coming in small portions from a large number of sensors. The data should be stored so as to facilitate efficient retrieval of a (possibly large) data array by sensor identifier and time interval, e.g. to draw a graph.

The system is described by three parameters: the number of sensors S, incoming data rate in megabytes per second A, and the storage period Y. In single-computer implementation there are limits on S, A, Y that can't be achieved without computer upgrade to non-commodity hardware which involves disproportional system cost increase. When the system design goals exceed the S, A, Y limits reachable by a single commodity computer, a viable solution is to move to distributed architecture — using multiple commodity computers to meet the design goals.

Approaching the distributed architecture design from scratch is not wise as there is a number of systems of this sort available [1]. Thus we turned to distributed NoSQL databases. Besides performance and capacity gains, the switch to distributed architecture promises to improve availability and fault-tolerance of the system.

We have chosen Riak [2] among NoSQL databases for the following reasons:

DOI: 10.25209/2079-3316-2017-8-3-61-85

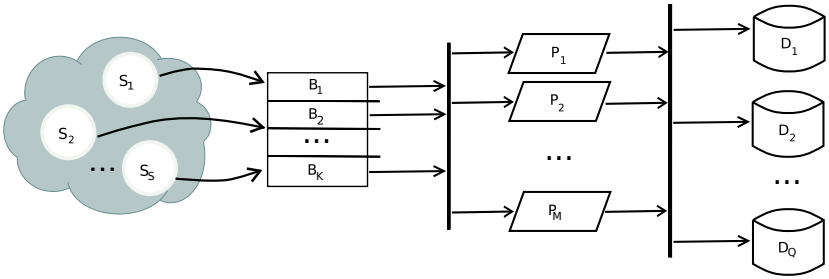


Figure 1: General sensor data storage model with buffering.

- Riak can function as an AP system in the sense of the CAP theorem. [3] This means the system sacrifices strict data consistency in favor of service availability in the face of partitioning (network or node failures). We think that a sensor data storage system needs to be highly available to minimize data loss;
- Riak is a fully decentralized design, all nodes are equal. Again, this is good for availability;
- Riak's data storage backend (named Bitcask [4]) is a log-structured store, writing only a small number of files to the underlying filesystem simultaneously, and as we know [5] this promises good write performance.

We consider sensor data storage implementation from the viewpoint of the general sensor data storage model (Fig. 1) introduced in [5], where the model is implemented using plain files in Linux OS ext4 and f2fs filesystems.

In a few words the model works like this. Data portions obtained from sensors $s_1 \dots s_S$ are grouped in the buffer layer $B_1 \dots B_K$. As buffers grow full, processes $P_1 \dots P_M$ write the data out to the secondary storage $D_1 \dots D_Q$.

Like in [5], in this paper we concentrate on write performance. We don't study read performance here, however we think about read efficiency when choosing the storage organization.

The rest of the paper is organized as follows. In section 1 we consider the general sensor data storage model implementation using Riak key-value store as the secondary storage $D_1 \dots D_Q$. In section 2 we show the problems we ran into and the changes we made to Riak to let it cope with the load characteristic for cyclic sensor data storage. In section 3 we consider the system performance and suggest modifications to Riak merge

machinery. In section 4 we test system performance and discuss the testing results. In section 5 we suggest possible further steps to performance increase, and in section 6 outline a Riak-based sensor data storage system architecture.

1. General storage model implementation based on Riak

The data in Riak are stored in the form of *objects*; every object is identified by a unique *key* and contains a *value*.

Riak objects differ from conventional files in that one cannot easily append data to objects: the objects are created with a value and then if you want to change the value you have to read the object, modify the value and store it in Riak again, which is not cheap. Therefore from the very beginning we are determined to use Riak objects in single assignment mode.

In the file-based general storage model implementation [5] we appended data to files in portions of B bytes; in Riak-based implementation we will create objects with value size of B bytes instead. The objects' keys will be based on sensor identifier; Riak provides fast access to objects by the keys.

In the file-based implementation, write performance (with constant volume of data in the system V) depended strongly on the number of sensors S as it translates directly to the number of files written simultaneously. In Riak-based implementation the write performance *does not depend* on S! All objects can belong to a single sensor, or every object can belong to its own sensor—there is no difference from Riak's perspective, it deals with a set of objects with user-supplied keys. Naturally, when growing S one has to reduce the data storage period Y to keep V constant.

1.1. The test routine

Riak properties were studied experimentally before. In [6] they study Riak reaction to changing the number of cluster nodes (both increasing and decreasing): scalability, availability and elasticity — the linearity of Riak response to node addition or removal. They use `basho-bench` [7] to generate the test load. In [8] the author studies latency, availability and consistency properties in connection with Riak configuration.

By contrast, we test not Riak per se, but Riak bundled with the sensor data storage system. We are interested in write performance expressed in Megabytes per second, whereas the studies mentioned above consider throughput as the number of Raik operations per second, saying nothing about Riak object sizes. For the test program we use a fragment

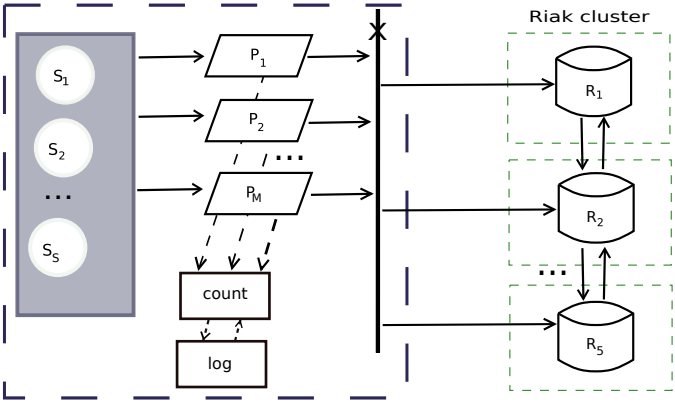


Figure 2: Test program structure

of the future sensor data storage system—the fragment responsible for writing data from memory to Riak, and feed it with test data stream. From test to test, we change the test program, Riak configuration and (if necessary) Riak sources to achieve the best write performance. When satisfied with the performance, we will implement the rest of the system by building up on the test program.

At first we tried to implement the test in C language using Riak C driver [9] in synchronous mode. We soon realized that single-threaded synchronous implementation does not perform well because of the round-trip delay in request handling. We thought to improve performance by using multiple requests concurrently using Riak C driver in asynchronous mode, but than decided to switch to Erlang language with Riak Erlang client [10]. In Erlang one can use lightweight processes to implement multiple outstanding requests cleanly, doing one synchronous request per process.

The test program structure is shown at Fig. 2. Writer processes $P_1..P_M$ send `put` requests to Riak nodes $R_1..R_N$ in turn. The value in the `put` requests is a random data block of size B, while the key is a 32-bit number where the lower 24 bits contain a sensor number $(1..S)$, the higher 8 bits contain the writer process identifier $(1..M)$. The processes P_i imitate arrival of data portions from sensors $S_1..S_S$ cyclically.

After every successful Riak `put` request the writer processes $P_1..P_M$ send a message to the `count` process which registers the volume of data written to Riak. Once a minute the `log` process requests the counter

Table 1: Riak node specification

CPU	Type	Intel(R) Core(TM) i5-4670
	Frequency	3400 MHz
	Cores	4
RAM (2 DIMMs)	Type	DDR3
	Volume	2048 MB x 2
	Frequency	1333 MHz
Operating system	Version	Debian 3.16.7
	Architecture	amd64
	Kernel version	3.16.0-4-amd64
Hard disk drive	Model	Seagate Barracuda ST3500630AS
	Volume	500 GB
	Rotation speed	7200 RPM
	Interface	SATA 3Gb/s
Ethernet Network interface	Model	Intel Ethernet Connection I217-V
	Driver	e1000e
	Standard	1000BASE-T

state from the `count` process and writes it to a logfile along with the timestamp; the `count` process then resets the counter, starting a new accounting period.

The relative volume of data V created in Riak during the test depends on the test parameters as follows:

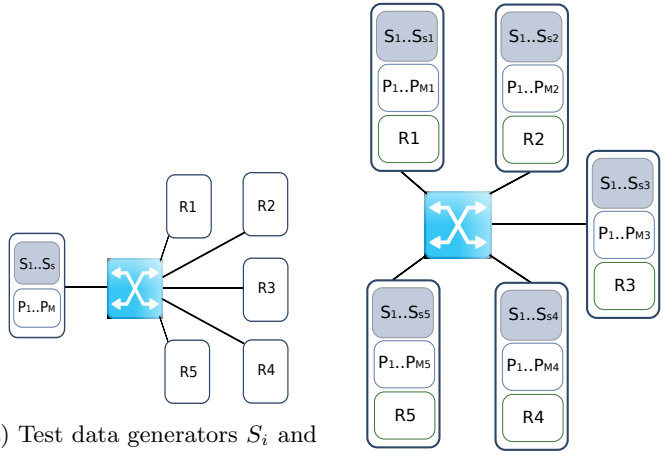
$$V = \frac{SBM}{C}, \tag{1}$$

where S is the number of sensors, B is the data buffer size equal to Riak object value size, M is the number of writer processes, C is the total filesystem volume of the whole Riak cluster, V is the planned volume of data in the system as a share of C.

When we start the test, the volume of data in Riak grows from zero to V and at this point stops growing no matter how long the test goes—because after this point the number of objects in Riak does not change, new `put` requests create objects with keys that already exist in Riak, making old objects obsolete ("dead" in Riak terms).

The space usage in Riak node filesystems where Bitcask stores its data ($\sum_{i=1}^{N} U_i$) can be much higher than V; we will consider it in detail later.

System configuration of Riak nodes is shown in the Table 1. We used two variants of the test setup. At first we had a single test data generator



(a) Test data generators S_i and Riak client processes P_j run on a front-end computer connected to Riak nodes $R_1 \ldots R_5$ using a Gigabit Ethernet switch

(b) Data generators S_i and Riak client processes P_j run on every Riak node

Figure 3: The test setup

running on a dedicated front-end computer (Fig. 3). As the performance grew in the course of the work, the network interface of the front-end computer became a bottleneck, and after working around that, the bottleneck shifted to network interfaces of Riak nodes. The interfaces carry not only new data sent from the front-end computer, but also Riak node-to-node traffic. At this point we decided on distributed data submission to the system (Section 6), and continued the testing with test setup variant (b) where every Riak node runs a test data generator, imitating data coming from external sources via an additional network interface.

Riak is configured with replication factor `n_val=1` so as to obtain the upper estimate of performance in the tests. Increasing the replication factor would lower the performance and increase the load on system resources—primarily on the disk space and the network bandwidth. We did no testing for these effects in this paper. The replication setup in Riak is so versatile, we decided to postpone the testing until we can chose the configuration according to an end user requirements.

When building the graphs presented below we often apply data smoothing with the "exponentially weighted moving average" filter [11].

The smoothing constant α is chosen individually for each graph as a trade-off between the visual appearance and detail display. The fact that EWMA filter has been applied is shown in the graph legend; we use EWMAn notation to denote the ewma filter with smoothing constant of $\alpha = 1/n$.

2. Riak misbehaviour in massive write test

The first experience from running the Erlang test with Riak was surprising. Soon after starting the test the disk usage in the filesystem underlying Riak (U) reaches 100% and then Riak locks up refusing to accept more data (the `put` request start returning `{error,enospc}`). This was unexpected because the data volume V calculated from experiment parameters was considerably lower than the filesystem size C.

We found two causes for this behaviour.

2.1. Cause1: put—merge disbalance

We run Riak with the default Bitcask [4] storage backend. Bitcask is a log-structured store: the data arriving in `put` requests are written to a single huge file. The file is rotated when it reaches a threshold which was set to 1 GB in these tests. An in-memory index called *keydir* provides mapping from keys to file positions where their respective data are stored.

When a `put` request contains a key already present in the store, the index is updated to point to the new data. The old data become obsolete ("dead" in Bitcask terms); they are still kept in a file but `get` requests for that key will read newer data instead. As more keys are updated, more old data become dead and it becomes necessary to reclaim the dead space, making it available for new data. The process of reclaiming dead space in Bitcask is called "merge".

The merge process runs in parallel with Bitcask API that handles `put` requests. The merge process operates on rotated files; it reads one or more files, filters out dead data and writes remaining live data to a new file, the merge result. The keydir is updated to reflect the data move. The files processed by merge are deleted and their blocks become available for new `put` requests.

The scheme works well as long as `put` requests are relatively rare. But when we flood Riak with `put` requests that set new values to existing keys, the merge process fails to keep up so the volume of dead data in Bitcask files grows. In the end, the merge process does not have enough filesystem space to write merged file so dead space cannot be reclaimed anymore. This is the lock-up situation: no space for merge to operate and no space for new data arriving in `put` requests.

2.2. Cause2: stale file descriptors

When the Bitcask merge process removes processed files, one can naturally expect the filesystem usage to decrease by the size of the removed files. But with Riak this sometimes (often) does not happen. We see the log line saying the file is merged and deleted, `ls` no longer sees the file but `df` shows steady filesystem usage increase—no decrease as one would expect after the deletion. Apparently the file is not actually deleted by the filesystem because it is held open by some process.

It turned out that Bitcask uses a file descriptor cache. Any descriptors used by `get` or `put` operations are kept for possible reuse. Obviously these descriptors should be closed and purged from the cache when the file is deleted, but doing so turned out difficult in current Bitcask implementation. Bitcask is implemented by a large number of light-weight Erlang processes. Merges are done by a separate process, and every Bitcask API client is also a separate process. The file descriptor cache is kept in private data (the "process dictionary") of the client processes. A file can be cached by multiple processes. Thus, when a file is deleted, we need to let all client processes know that the file is gone and need to be purged from the cache. Unfortunately, there is no easy way to do that.

We have found mentions of this problem dated back to 2013 [**12**, **13**] but the problem still exists in contemporary Riak code (Riak KV 2.1.4). Probably most Riak users do not notice this problem because their Riak stores are large, files are numerous and `get` requests naturally rotate descriptors in the cache so the problem is mitigated. In our test configuration the store is relatively small (only 100GB per node), with 1GB rotation limit this yields as little as 100 files overall—they can all fit in the cache and stick there forever, so the problem manifests itself loudly.

2.3. Remedy to put—merge disbalance

To avoid the lock-up described above one can lower the rate of `put` request when the filesystem usage gets close to 100%, leaving more I/O and CPU resources to the merge process. Implementing rate control in the test program is not an option, because in distributed configuration the test program cannot know disk usage situation on the Riak nodes where its `put` requests are eventually handled. We decided to implement flow control using the fact that the clients are numerous but synchronous: they don't send another `put` request until the one already sent is finished. Thus all we have to do is to introduce a variable delay in `put` request handling.

To this purpose we define two thresholds of filesystem usage U:

U_l lower threshold; at this point and below, the delay is zero

U_h higher threshold: at this point the delay is as large as 1 second.

We calculate the throttling delay T based on U as follows:

$$T = \max\left(0, \frac{U - U_l}{U_h - U_l}\right)$$

We have set up throttling with $U_l = 95\%$ and $U_h = 100\%$ and now the merge process keeps up with dead data deletion so we observe lock-ups no more.

2.4. Remedy to stale file descriptors

So when the merge process removes a file, we need to let client processes know that the file is gone and need to be purged from the cache.

One can think of an event scheme: the merge process would broadcast a message "file X is deleted" to client processes. But there is no such thing as "broadcast message" in Erlang (and for a good reason). We would need to maintain a list of client Pids in the merge process; this is inconvenient but doable. Then, a client process is not part of Bitcask: it is a process from Riak that called a function from Bitcask API. Even if it is sent a message, it will only be able to react on it (delete the file) when the client process performs another *put* or other Bitcask API operation—that is, we get a generally unpredictable delay. Further, the process might have its own message handling code outside Bitcask which can be confused by unfamiliar message. In short, the message idea is unfeasible.

The right way to do this is probably to get rid of multiple descriptor caches in client processes, implement common cache as a named ETS table accessible to the merge process (for descriptor purging) and to the client processes (for descriptor reuse).

For the purposes of this study, we implemented a quick workaround: let every client process inspect its cache periodically, checking existence of every file using `stat` system call. This rude workaround is good enough for our purposes because in our test every client process enters Bitcask often, hence the time interval from file removal to cache entry purging is never much longer than the inspection period.

3. Merge efficiency

With changes from Section 2 in place the test no longer crashes and we can study the system performance.

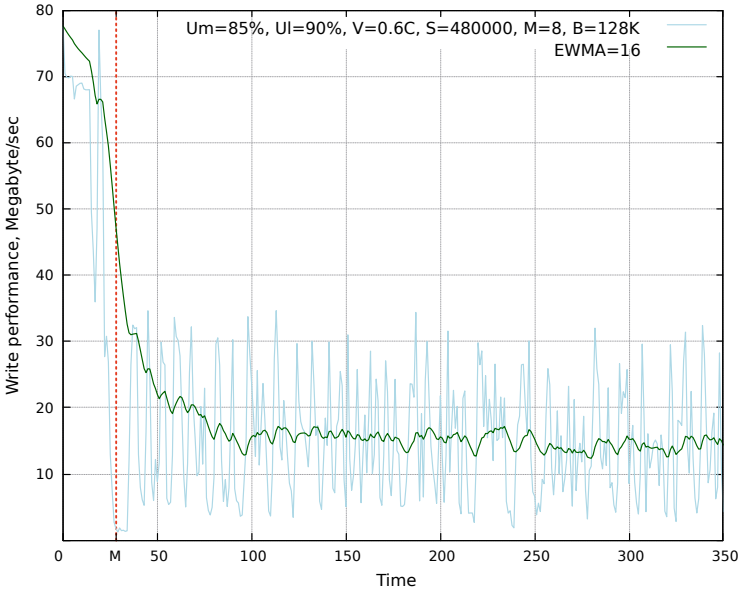


Figure 4: Performance variation during a test. The point M is where the merge activates and we see the performance drop

Looking at write speed change at Fig. 4 we observe decent speed in the beginning and then it drops abruptly when the merge activates. In the beginning the filesystem usage U is low, the merge is not required, the filesystem only gets write requests to Bitcask files which are few. When the merge activates, the filesystem starts getting read requests which compete with write requests which lowers the performance considerably.

Our experience with Riak suggests that Riak developers primarily targeted applications where the data retrieval requests (`get`) prevail while data deletion (`delete`) or update (`put`) requests are relatively rare so the need for merge arises unfrequently. Riak can even be configured to do merge operations in off-peak hours (`bitcask.merge.window` [14]).

In the cyclic sensor data storage application the situation is quite the opposite. We have an intensive flow of `put` requests that update data for existing keys so the need for merge exists nearly always. This way, to increase system performance we have to improve merge efficiency.

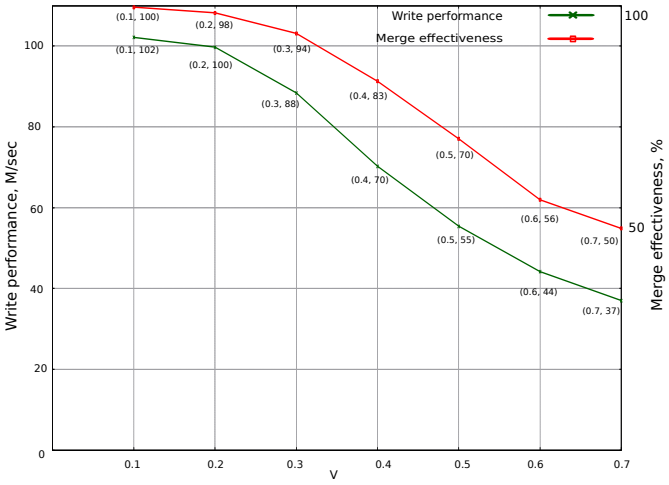


Figure 5: Write performance and average dead data share in merged files by relative volume of live data in the system V

3.0.1. *Merge queue control*

The merge in Riak is implemented by several Erlang processes. In every *vnode*[1] a dedicated Erlang process monitors the rotated Bitcask files and decides which files will participate in the nearest merge based on the configured bitcask merge policy [14]. The list of participant files is sent to the merge queue. The queue is run by a merge process; there is exactly one merge process per physical node. The merge process handles the queued files in FIFO order; it reads the file through, skipping dead data and copying live data to a new file—the merge result.

The files in the queue belong to different vnodes and differ by dead data ratio D. The merge efficiency must be proportional to D: the larger the dead data share, the less live data are there to copy, the more space will be freed by the merge. Indeed, at Fig. 5 one can observe distinct

[1]vnode in Riak—a short for "virtual node" responsible for storing data with keys whose hash values fall in certain range. The number of vnodes in the system is a Riak configuration parameter and is normally several times higher than the number of physical nodes. Vnodes are distributed (nearly) evenly over the physical nodes and can be redistributed when physical nodes are added of removed from the system

correlation between the system performance and the average share of dead data in the files committed to merge. This way, we could improve merge efficiency by smarter queue handling—processing the files with higher D first.

3.0.2. *Merge trigger based on filesystem usage U*

Riak merge policy configuration allows one to specify when the merge should start based on file fragmentation threshold, either key- or data-wise. It turns out that in write-intensive applications you cannot configure merge trigger properly with these parameters. If you set the threshold high, you risk to run out of filesystem space. If you set the threshold low, the merge will start too early and needlessly consume I/O bandwidth competing with `put` activity. We would like the merge process to maintain U at certain level, say 90%, but achieving that is not possible with static fragmentation threshold setting.

With time, the share of dead data D in Bitcask files only grows, so the later we start merging, the higher merge efficiency we obtain. This suggests that one shouldn't start merging at all until U is above the configured threshold. Riak could track filesystem usage U using `df` utility or `statfs` system call and trigger the merge only when it is necessary to free up some disk space.

3.0.3. *Quick merge for 100% dead files*

The files with all data dead ($D = 1$) are an important special case. These files do not need merge proper: no need to read the file for the remaining live data, the file can be simply deleted. This is much more efficient than the regular merge procedure.

3.0.4. *The Fisherman algorithm*

We will re-implement merge control using an algorithm which is similar to that of a fisherman who fishes to satisfy the hunger. When the fisherman is not hungry, he lets small fish go, taking only big fish. If no big fish appears for a long time, the fisherman gets hungry and starts taking smaller (medium) fish. If the medium fish is plentiful enough to sate, the fisherman continues to catch and eat the middle fish, still letting the small fish go. If there is not enough middle fish, the fisherman

lowers sights further, taking smaller and smaller fish. With time, the fisherman adapts to the pond he fishes in: he knows the size of fish to take or let go to always be satisfied without eating too small fish.

D will denote the fish size $D = 1$ meaning the biggest fish, $D = 0$ the smallest fish. D_m is the "take fish" size threshold that the fisherman finds by experience. The fisherman never lets go the biggest fish ($D = 1$), even if he is not hungry at all—but biggest fish is not caught frequently enough for this rule to cause overeating.

When applying the Fisherman algorithm to merge queue control, the fish size corresponds to the share of dead data D in a queued file. The hunger is expressed as $H = (U - U_m)/(1 - U_m)$, where U is the current filesystem usage level, U_m is the target level that the algorithm should sustain.

A queued file is selected for merge when $D >= D_m$, where

$$D_m = \begin{cases} \min(1 - H, \mathit{ewma}(D_i)), & H > 0 \\ 1, & H \leq 0. \end{cases}$$

Here $\mathit{ewma}(D_i)$, an exponentially weighted moving average of all the fish eaten, stands for fisherman's memory of what fish is eatable. D_m depends on the hunger H also, which provides for learning: if the hunger made the fisherman take an unusually smaller fish, ewma will lower and in the future the fisherman will take smaller fish even if not too hungry. On the other hand, if an unusually big fish is caught, ewma will rise so the fisherman will not take small fish in the hopes for more big fish in the near future.

4. Testing

The parameters of all experiments we do are tied by 1 (page 65). We run the same test program with different parameter values to study the system behaviour.

4.1. Write performance by live data share V

In the first place we would like to know if the measures taken in Section 3 helped to lessen the performance drop with merge activation. We

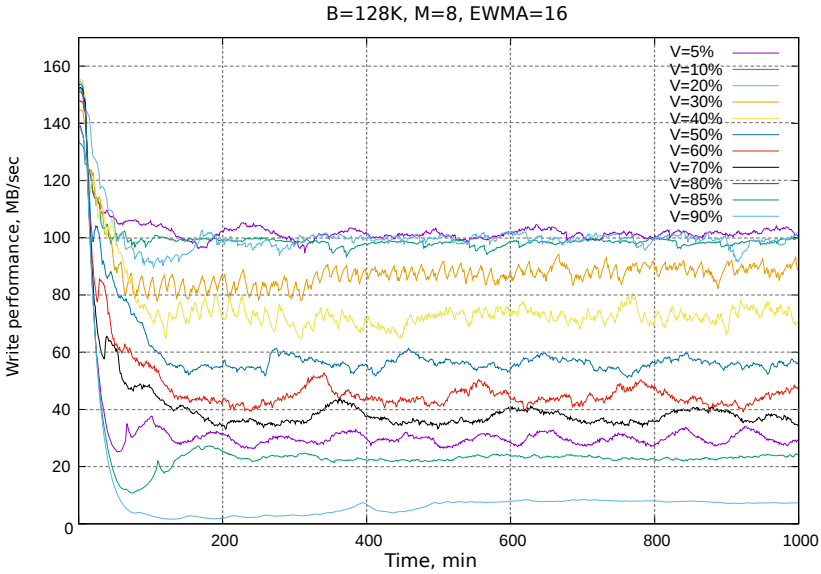


Figure 6: Write performance change during the test at different levels of live data in the store V

have repeated the tests shown at Fig. 4, 5, the results are shown at Fig. 6. We varied V by changing the number of sensors S. The tests were done with a single Riak node ($N = 1$): there is no sense to use multi-node setup at this point as the merge is performed autonomously by every node.

The performance drop became smaller indeed as compared to Fig. 4. The improvement is the most clear in tests with lower V because in these tests there are many files with $D = 1$ in the merge queue and these are processed efficiently thanks to optimization in Section 3.0.3.

In the test with $V = 0.9$ the system works in C area (Fig. 7) where files being merged contain little dead data (D is close to zero), the merge efficiency is low and the system has to throttle `put` requests (Section 2.3). Now that we improved the merge efficiency, the system does not have to resort to throttling with $V <= 0.8$, which can be considered an achievement.

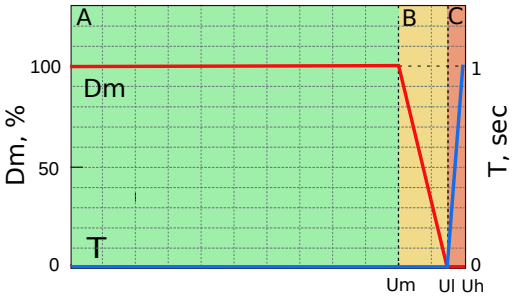


Figure 7: A area—efficient merge, B area—less efficient merge ("conventional mode"), C area—merge is slower than flow of new data, `put` requests throttle activated

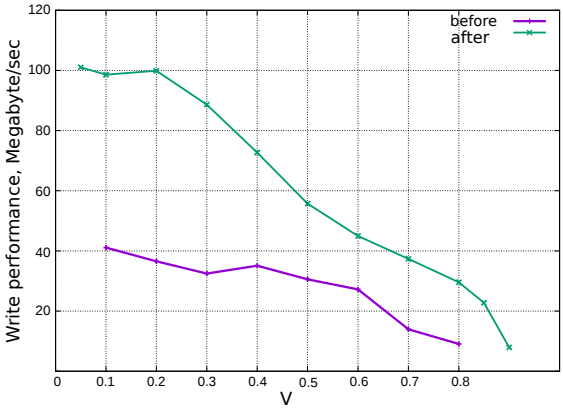


Figure 8: Settled write performance by live data share V, before and after Section 3 optimizations

Fig. 8 summarizes test results before and after optimization to demonstrate the optimization effect.

4.2. Write performance vs. Bitcask rotation size R

The best performance so far is achieved in "A" area (Fig. 7), where the need for disk space is satisfied purely by merging files with $D = 1$, so the merge process consumes almost no resources thanks to Section 3.0.3 optimization. This operation mode resembles a boat moving in planing

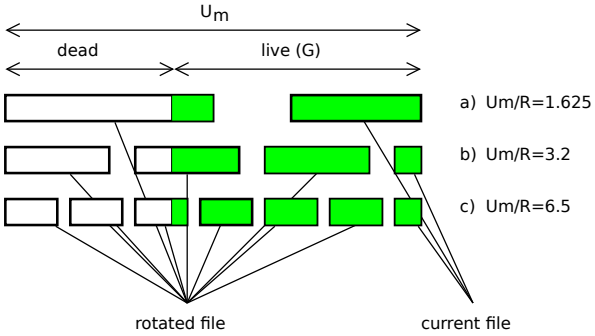


Figure 9: Distribution of live data in Bitcask files at the moment $U = U_m$ at three different Bitcask file rotation sizes.

mode when the water resistance to the move drops drastically. We will use the term "planing" to denote the system working in "A" area.

In Section 4.1 tests we achieved planing by lowering the live data volume V which also has the negative effect of lowering the available storage space. Could we achieve planing without lowering V? For that we would need to have a mergeable file with $D = 1$ in at least one vnode of the physical node where U have reached U_m threshold.

The average number of Bitcask files per vnode at the moment $U = U_m$ is given by

$$F = \frac{C_i}{n_i}\frac{U_m}{R}, \tag{2}$$

where R is the Bitcask rotation size (`bitcask.max_file_size`), C_i is the filesystem capacity on the physical node number i, n_i is the vnode count on the physical node. The specifics of our test is that all Riak objects created by it have equal lifetime: the earlier an object is created or updated, the sooner its data become dead when replaced with new data. This way, live and dead data distribute across Bitcask files as shown on Fig. 9.

From Fig. 9 we can construct the planing condition:

$$G \le (F - 1)R, \tag{3}$$

where $G = V\frac{C_i}{n_i}$ is the volume of live data in one vnode. By substituting F and G to (3) we obtain the planing condition in the R by V form:

$$R \le \frac{C_i}{n_i}(U_m - V). \tag{4}$$

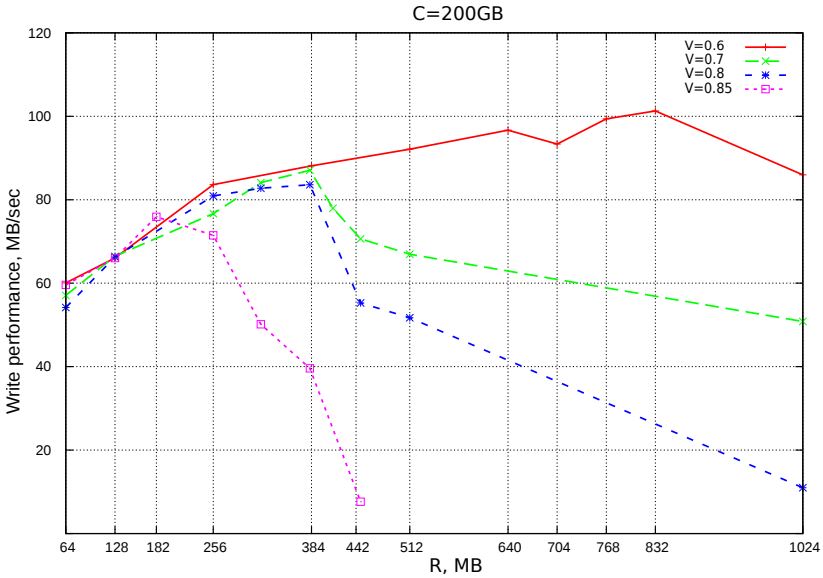


Figure 10: Write performance vs. Bitcask file rotation threshold R with different node filesystem capacity C_i and relative data volume V.

Relation (4) is only approximate as it does not account for non-uniform data distribution among vnodes, space taken by auxiliary Bitcask files, and filesystem overhead. Still it is usable as a reference point for Riak configuration. One should also take into account that the number of vnodes on a physical node n_i can change when the system gets partitioned.

From (4) we see the system will remain planing when increasing V if we decrease R proportionally. However when decreasing R we will have more frequent file rotations which adversely affects performance (Fig. 10).

The tests shown at Fig. 8 were done at $C_i = 100\text{GB}$, $R = 1\text{GB}$, $n_i = 64$, $U_m = 0.8$. This is not the best possible configuration: the planing condition is true only for $V \leq 0.16$. In reality the test at $V = 0.2$ still demonstrates the maximum performance, it gets worse only at $V = 0.3$. According to the system log, at $V = 0.2$ the system is still planing, and only at $V = 0.3$ it is not planing anymore. We have told that (4) is only approximate and indeed observe a small bias to the lower side, probably caused by nonuniform data distribution over vnodes. The condition (4) is formulated for the worst case where files in all vnodes grow evenly and

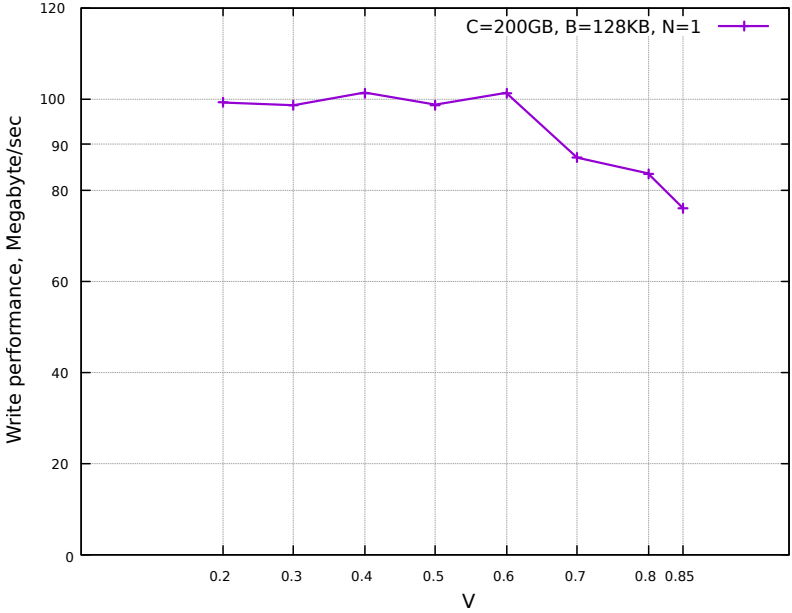


Figure 11: Write performance vs. data volume V in planing mode

at $U = U_m$ all vnodes suggest merge candidates with the same D. When the distribution is non-uniform, some vnodes can suggest candidates with $D = 1$ even if average D among all candidates is much less than 1. Even one candidate with $D = 1$ is sufficient to sustain planing.

4.3. Write performance vs. data volume V when planing

The purpose of the test is to find a balance between performance and disk space efficiency. These tests differ from the test in Section 4.1 by using optimal Bitcask rotation threshold R for each test according to (4), so in all tests the system was planing. The test results are shown at Fig. 11. We believe the performance drop at $V > 0.6$ is due to filesystem fragmentation. When increasing V, filesystem usage U grows as well, increasing probability of file fragmentation which in turn can reduce the filesystem write performance.

When configuring the system one should take into account that V will increase in case of system partitioning. For example, in a five-node cluster configured to work at $V = 0.6$, a node failure will case the relative data

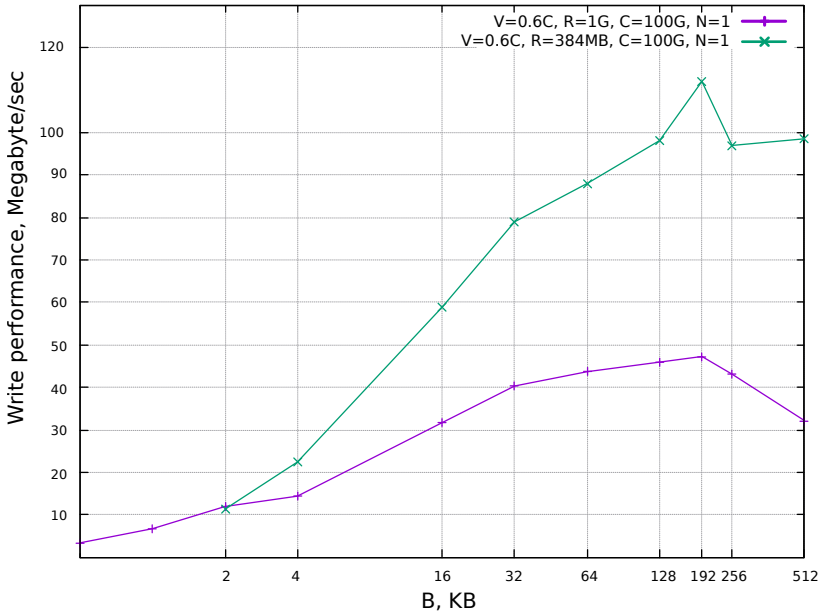


Figure 12: Write performance vs. Riak object size B

volume increase to $V = 0.75$ which will aggravate the impact of the failure on performance. With one node of five failing the performance drop will be not 20% as one would expect, but 35%.

4.4. Write performance vs. Riak object size B

The purpose of the test is to find how the write performance depends on Riak object size. Riak documentation [**15**] recommends "that Riak objects stay smaller than 1-2 MB and preferably below 100 KB". However it says nothing on performance with smaller object sizes like 1...4KB.

As the performance is strongly affected by data volume V (Section 4.1), in this test series we will maintain constant $V = 0.6$. To achieve this, when changing B we also change the number of sensors S — that is, when reducing B we decrease the size of Riak objects but increase their number. The test results are shown at Fig. 12.

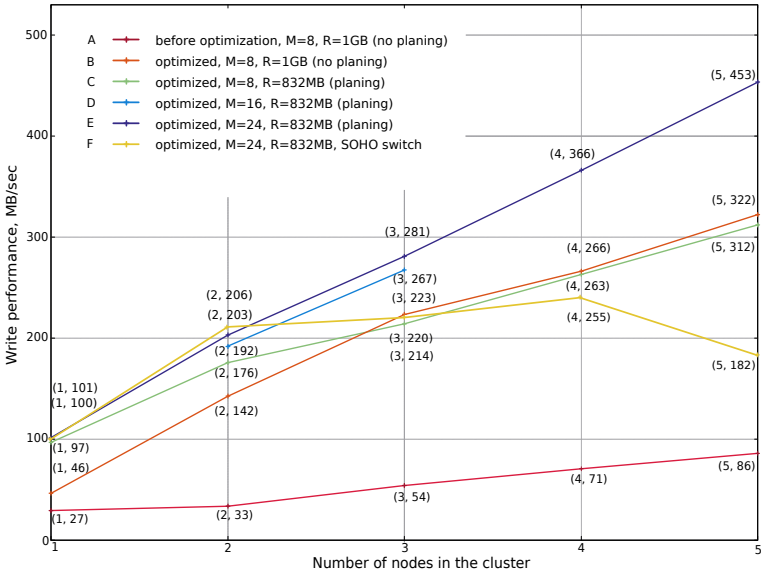


Figure 13: Write performance vs. the number of Riak nodes

4.5. Write performance vs. the number of Riak nodes N

The tests has been conducted for $N \in \{1, 2, 3, 4, 5\}$ at B=128K. The results are shown at Fig. 13.

Line A at Fig. 13 has been obtained prior to merge optimization from Section 3, so the results are relatively low. Line B has been obtained with merge optimization but at suboptimal Bitcask file rotation threshold R (so no planing). Line C has been obtained with merge optimization and optimal R that provides planing mode.

Line C results turned out lower than expected. We we able to improve them by increasing M—the number of Erlang processes issuing Riak `put` requests on every Riak node. The results with increased M are given by lines D and E—these are the best results achieved by now.

We could explain why increasing M was necessary to improve the results as follows. Objects are distributed among Riak nodes according to hashed key value, regardless of which node is handling the `put` request. As the number of Riak nodes increase, the share of objects that are destined

to a node other than the node handling the `put` request increases as $1 - \frac{1}{N}$. The average request handling time increases accordingly, and we have to increase the number of concurrent requests to maintain throughput.

Line F has been obtained at the same parameters as line E, but using a SOHO Gigabit Ethernet switch with smaller buffer memory (1 Mbit). We have noticed that in the tests with $N > 2$ the switch started transmitting IEEE 802.3x Pause Frames during the test, throttling the network. This had a huge impact on the system performance, up to negative growth in 5-node test.[2]

As the experience with lines C and F shows, when scaling the system up it is always desirable to test the obtained performance increase and, if it is lower than expected, look for bottlenecks.

5. Further merge optimization possibilities

In addition to the measures taken in Section 3, we see some more possible steps which we have neither implemented nor tested yet.

5.1. Selective read

This optimization targets the system working in B area (Fig. 7). Even if the system was initially tuned for planing (area A), it can drop to area B in case of partitioning.

As we told in Section 2.1 above, the Bitcask merge process works by reading fragmented files, filtering out dead data and writing live data to the resulting file. The filtering works by looking up the key in the in-memory *keydir* to see if the keydir file position points to the place where the merge process have found the entry. If yes, the entry is live; if not, the entry is dead.

The less live data is in the file, the less justified is reading the file through. If live data in the file are sparse, we would be better off by picking live data just like `get` does, and then remove the file without the useless dead data reading, like in planing mode.

[2]The problem is known as "TCP incast" [16]. At first it caused the test collapse before reaching steady state. We were able to mitigate the problem partially by disabling network interface TSO (TCP segment offload) for line F tests with 3,4, and 5 nodes. Disabling TSO reduces bursts of packets addressed to the same node that overload the switch buffer memory and cause the problem

To implement this approach one would need to be able to walk over all live keys in certain file efficiently. This will probably require a keydir modification or maintaining an additional index. Whether it is justified is an open question.

5.2. Write buffering

Tracing Bitcask system calls shows that the write length is variable and determined by the key and data sizes originally supplied by the user `put` request—both for main stream and merge writes. That is, Bitcask generally writes data in blocks that are neither multiple of 4KB nor aligned to 4KB boundary. We know from [5] that writing in blocks smaller than the filesystem block size results in degraded performance, even if this effect was not very pronounced when only a small number of files are written simultaneously.

Increasing the write length by increasing the write buffer size B would result in proportional increase of the RAM volume required to implement the buffer layer of the general sensor data storage model. But we can also increase the write length without changing B—by implementing additional data buffering prior to file writing, similar to the buffering implemented in C stdio library. The size of the buffer can be chosen based on benchmarking results to provide the best performance on the target platform.

Implementing the buffering will not consume much memory: one buffer per open output file, the number of files opened by Bitcask for writing simultaneously being very moderate (2 files per vnode).

6. Sensor data storage architecture

As the tests show, for a large number of sensors Riak provides much better write performance as compared to plain filesystem-based implementation [5]. Besides, Riak allows to increase the system performance and capacity even further by increasing the number of nodes in the system. In our tests even a 2-node system performance exceeds the Gigabit Ethernet network throughput. This way, the first test installation variant (Fig. 3) does not scale, and the same is true for a sensor data storage system with a single front-end computer receiving all sensor data. A scalable system must have multiple sensor data collection points.

A natural solution for maximum availability is to open a sensor data collection port at every Riak node. The application servicing the port will receive and buffer data in accordance with the general model (Fig. 1) and pass the data from complete buffers on to Riak instance on the local node. Riak node will forward the data to one or more nodes in accordance to hashed key value and the replication policy. The application will use a protocol with sensor data sources to optimize and balance load on the nodes. The application instances on the nodes will interact to detect system partitioning (node or network failure), adapt to work in partitioned mode and return to normal mode when the partitioning is over. Implementation details are the subject of a separate article.

7. Conclusion

We have evaluated general sensor data storage model [5] implementation using distributed key-value store Riak KV for the secondary storage.

The role of cyclical sensor data store turned out hard for Riak. At first the system locked up under test in a half-hour and we had to do a couple of fixes to Riak to get the test running continuously. Then we addressed the system write performance and have found it suffering from inefficient merge (a kind of garbage collection) in Bitcask subsystem of Riak. We have implemented an enhanced merge control algorithm and achieved a fivefold increase in the write performance for our load type. We have tested the system scaling from one to five Riak nodes and observed close to linear performance growth by 85 Mbytes/s per node added. We have tested system write performance while varying relative data volume, Riak object size and Bitcask file rotation threshold, and built characteristics useful for application development.

When planing a real system one should consider repeating the tests on the computing platform of choice as quantitative results differ wildly from platform to platform and bottlenecks lurk everywhere.

In our experience Riak and Erlang turned out rewarding subjects. We would like to thank creators of Perl [17] programming language and redbug [18] trace utility that we used to study the system operation, creators of gnuplot [19] which has built all graphs in this paper, and also the countless people who made Debian GNU/Linux [20] possible and keep it in existence.

References

[1] K. Grolinger, W. A. Higashino, A. Tiwari, M. A. M. Capretz. "Data management in cloud environments: NoSQL and NewSQL data stores", *Journal of Cloud Computing: Advances, Systems and Applications*, **2** (2013), 22. ↑ 61

[2] *Riak: a decentralized datastore*, Basho Technologies, URL: https://github.com/basho/riak ↑ 61

[3] E. A. Brewer. "Towards robust distributed systems", Nineteenth annual ACM symposium on Principles of Distributed Computing (Portland, Oregon, United States, July 16–19, 2000), pp. 7, URL: https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf ↑ 62

[4] J. Sheehy, D. Smith. *Bitcask: a log-structured hash table for fast key/value data*, Basho Technologies, 2010-04-27, URL: http://basho.com/wp-content/uploads/2015/05/bitcask-intro.pdf ↑ 62,67

[5] N. S. Zhivchikova, Yu. V. Shevchuk. "Experiments with sensor data storage performance", *Program Systems: Theory and Applications*, **8**:4(35) (2017) (to appear). ↑ 62,63,82,83

[6] A. Ghaffari, N. Chechina, P. Trinder, J. Meredith. "Scalable persistent storage for Erlang", *Twelfth ACM SIGPLAN Workshop on Erlang* (Boston, MA, USA, September 25–27, 2013), pp. 73–74, URL: http://eprints.gla.ac.uk/107445/1/107445.pdf ↑ 63

[7] *Basho Bench, a benchmarking tool*, Basho Technologies, URL: https://github.com/basho/basho_bench ↑ 63

[8] Z. Zatrochová. *Analysis and testing of distributed NoSQL datastore Riak*, Master thesis, Masaryk University, Brno, 2015, URL: https://is.muni.cz/th/374482/fi_m/thesis.pdf ↑ 63

[9] P. Nosek. *C client for Riak — NoSQL database*, URL: https://github.com/fenek/riak-c-driver ↑ 64

[10] *Riak Erlang Client*, Basho Technologies, URL: https://github.com/basho/riak-erlang-client ↑ 64

[11] *NIST/SEMATECH e-Handbook of Statistical Methods*, URL: http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc431.htm ↑ 66

[12] Github Basho Bitcask, URL: https://github.com/basho/bitcask/issues/113 ↑ 68

[13] Github Basho Bitcask, URL: https://github.com/basho/bitcask/issues/114 ↑ 68

[14] Basho Technologies, URL: http://docs.basho.com/riak/kv/2.1.4/setup/planning/backend/bitcask/ ↑ 70,71

[15] Basho Technologies, URL: http://docs.basho.com/riak/kv/2.1.4/developing/data-modeling/#sensor-data ↑ 79

[16] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, S. Seshan. "Measurement and analysis of TCP throughput collapse in cluster-based storage systems", *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08 (San Jose, CA, USA, February 26–29, 2008), 14 p., URL: http://www.pdl.cmu.edu/PDL-FTP/Storage/FASTIncast.pdf ↑ 81

[17] *The Perl Programming Language*, URL: https://www.perl.org ↑ 83

[18] *Erlang performance and debugging tools*, URL: https://github.com/massemanet/eper/blob/master/doc/redbug.txt ↑ 83

[19] gnuplot homepage, URL: http://www.gnuplot.info ↑ 83

[20] debian: the universal operating system, URL: http://www.debian.org/ ↑ 83

Submitted by *prof. Sergej Znamenskij*

Sample citation of this publication:

N. S. Zhivchikova, Y. V. Shevchuk. "Riak KV performance in sensor data storage application", *Program systems: Theory and applications*, 2017, **8**:3(34), pp. 61–85. URL: http://psta.psiras.ru/read/psta2017_3_61-85.pdf

About the authors:

Nadezhda Sergeevna Zhivchikova

Research engineer of Research Center for Multiprocessor Systems of Ailamazyan Program Systems Institute of Russian Academy of Sciences

e-mail: ming@pereslavl.ru

Yury Vladimirovich Shevchuk

Head of telecommunication laboratory of Research Center for Multiprocessor Systems of Ailamazyan Program Systems Institute of Russian Academy of Sciences

e-mail: sizif@botik.ru

The same in Russian: DOI 10.25209/2079-3316-2017-8-3-31-60